State-Space Covering Strategies for Guaranteed Proper Termination

Laurence R. Phillips, Senior Scientist
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-0977
lrphill@sandia.gov

Stephen J. Bespalko, Principal Investigator
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-0977
sjbespa@sandia.gov

Alexander Sindt, Technical Intern
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-0977
adsindt@sandia.gov

1. Background

Laurence R. Phillips has been conducting research into representational strategies, design and construction of large-scale information systems, and automated failure recognition and recovery for over twenty years. He has worked in the areas of geometric modeling and reasoning, military operations planning, and automated representation of complex information structures. His most recent work was the development of automated interpreters to convert HTML documents into complex objects and to render complex objects into HTML.
Stephen Bespalko has been involved in the design and implementation of large-scale real-time systems for over twenty years. For the past five years Mr. Bespalko has conducted in primary research into the problems confronting the implementors of large-scale systems at Sandia National Laboratories. His current research is in formal methods applied to satellite ground station components. Prior to joining Sandia National Laboratories, he spent 8 years as part of the R&D staff at Xerox Corporation. While at Xerox, Mr. Bespalko successfully applied formal language theory to the design of numerous high-tech components including a digital scanner, document pagination algorithms, graphics algorithms, fax devices, and communications protocols. Mr. Bespalko holds several US and International Patents resulting from his work at Xerox.
Alexander Sindt will be a Junior at the Massachusetts Institute of Technology where he is double-majoring in Computer Science and Philosophy. He has spent the last two summers working as a Student Intern at Sandia National Laboratories in the areas of GPS applications and Formal Languages.

2. Position

Complex systems that control real-world events are common, but confirming that such a system will always behave in an expected way is very difficult. We define a system to have *mission surety* if and only if system behavior is (a) syntactically complete (the system can actually perform every expected behavior) and (b) semantically correct (every input to the system results in an expected behavior). Current development methods attempt to demonstrate mission surety through extensive testing via simulation. These efforts do not always prevent surprises (e.g., discovery by other means

of incorrect code not exposed by testing) that lead to higher costs and missed schedules.
There are three principle tactics for achieving mission surety:

* Reuse existing code;
* Automate complex development tasks; and
* Design for failure detection and recovery.

Off-the-shelf (OTS) software can often be used to perform common system functions such as display management, network operation, database management, and so on. Because it has already been tested to some degree, OTS technology is expected to fail less often than equivalent custom software, and using it simplifies the development process. Although OTS components can improve the probability that a system will exhibit mission surety, they cannot in and of themselves guarantee it. In fact, using OTS components can complicate demonstrating that system behavior is complete because of their nominal "black box" status. Although specifying OTS components in a formal environment is of considerable interest to us, this is a digression from our current topic.
Moreover, there is no OTS technology for many relatively common mission-critical functions because of their idiosyncratic and context-sensitive nature, e.g., splitting bulk data into pieces, validating data sequence and structure, extracting meaning from data, and automatically detecting and recovering from runtime errors. Mission-compromising errors in these operations are common. Automating the development and implementation of these functions is a key mechanism for generating more capable systems at lower cost with greater mission surety. The essential means by which we can construct such automation is a context-sensitive grammar for pairing system state with system behavior. Behavioral rules can then be compiled (i.e., operational code can be automatically generated) as contextual dependencies are revealed during development.
The third strategy, failure tolerance, rests on two bases that devolve directly from mission surety:

* Complete representation of the system state space (assumed to be correct) and
* System behavior specification based on this system state representation.

In particular, the first assures us that it is possible to assign a response to every possible state and the second allows the system to function as desired in response to each state. The implication for fault tolerance is that some states are error states to which recovery behavior can be assigned. As side affects, a complete representation of the state space enables both the construction of a vocabulary for the system’s behavioral semantics and the precise delineation of viewpoints.

2.1 Context Sensitivity and State Space

We focus here on categorization of the state space into a covering set and its effects on

* Automating development of system specification using a context-sensitive grammar and
* Error detection and recovery.

The goal of modeling the system state space is to develop a set of system states (or state sets) that corresponds one-to-one with its behaviors. A tactic is to begin with the behaviors, assuming that there is at least one state per behavior. In a way, this is antithetical to automating the process, since examining behaviors will lead to a process that’s idiosyncratic and, thus, not generally specifiable. At any rate, achieving useful results by this tactic becomes hard when a large number of system states (indicated by a large number of behaviors) overwhelms the design process, which is ordinarily mostly manual. Systems designed for the real world are normally larger and more complex than their laboratory counterparts because the inherent variability of their input regimes implies a much larger behavioral range.
If we want our designs to be scalable, we must find a semantically meaningful categorization of all of the system’s behaviors such that all the behaviors in each category result from a single state and system states map onto the categories one-to-one. When this can be done, scaling a program into the real world won’t involve meta-level redesign because the number of states won’t change.
For example, suppose we are accumulating values in a buffer that may overflow. Naming a system state after every possible buffer value would be overwhelming. The astute developer will see that the system has only three *behaviors* of interest, regardless of the number of literal buffer values, because we essentially have three states: Accumulation (input and content manipulation), Output,

and Overflow. The task is to discover a grouping of buffer values into three categories such that executing the same system behavior against every value in each category is semantically meaningful. This is not always possible in the context-sensitive case because the dependence of behavior selection on the results of some or even on all previous behavior causes a combinatorial explosion of potentially meaningful states.

2.3 Meta-Design: Two Examples; One Formal Description

We address a problem we'll refer to as *decomposition*. This is a common task we refer to above as "splitting bulk data into pieces." Suppose we have a composite object of some kind and our goal is to break it apart into its components. When the boundaries are unambiguous, this operation is known as *parsing* and is fairly commonplace, although always idiosyncratic to the grammar at hand. Our basic scenario, familiar and fairly simple, is based on actual work done in handling data from a satellite system:

[startVal][count][frame1][frame2]…[frame(count)][endVal]

The grammar is simple, but in practice building the parser manually has led to some errors. Consider the following BNF fragment:

```
frames::= [startVal] frameList [endVal]
frameList::=[count] {basicFrame | basicFrame frameList}
basicFrame ::=...
```

The source of error is that the count is essentially ignored. Furthermore, actions for the first frame are different that the actions for the rest. The following fragment attempts to be sensitive to the context, but uses a context-free grammar.

```
frames::= frameList
frameList::=[startVal][1] basicFrame  |
                [startVal][2] basicFrame basicFrame [endVal] |
                [startVal][3] basicFrame basicFrame basicFrame [endVal]
               ...
basicFrame ::=
```

Now we have a special case for each value of count, and the action for processing a frame is replicated

$1+2+3+...+n = n*(n+1)/2$

times. The grammar has accomplished its mission, but in engineering terms we have greatly increased the odds that some frame won't be processed correctly, even for small n, because of the replication. We implemented an alternative grammar using an extension of the standard context-sensitive grammar (i.e., we removed the limitation imposed by Chomsky level 2 that the length of the LHS be less than or equal to the length of the RHS). The following BNF fragment is compact enough for human verification and allows the implementor to specify the semantic processing for the basicFrame in exactly one location:

```
frameList ::= {basicFrame | basicFrame frameList}
basicFrame::= [startVal][1]basicFrame[endVal]
basicFrame[startVal][n-1]frameList::= [startVal][n]basicFrame frameList
basicFrame ::=...
```

Even though this is a small generalization of a simple context-sensitive grammar, the impact on the architecture of the software component studied was enormous: rather than a huge number of 'special cases', the process could be specified with just a few dozen compact and precise rules.

Now suppose that the boundaries between the components are ambiguous, but that we are able to recognize a well-formed component when we acquire one. Notice that the grammar doesn't change, but we may now no longer reliably execute a production. Furthermore, the selection of a boundary not only determines the value of the putative basicFrame in question but also affects the range of possible values for the one that follows it and potentially all the rest as well.
Further suppose that we have no 'count,' i.e., we don't know how many frames are in the frameList. This situation occurred in a spatial reasoning problem to decompose a complex geometric shape into primitives.
We have the following partial grammar:

frameList ::= {basicFrame | basicFrame frameList}
basicFrame::= [startVal]basicFrame[endVal]
basicFrame[startVal]frameList::= [startVal]basicFrame frameList
basicFrame ::=...

The grammar is simpler, but implementing a process to decompose this structure is significantly more difficult. The primary issue here is that when the boundary between components is ambiguous, we cannot know whether the implementation of a production will result in a basicFrame or not. Furthermore, we cannot predict how many basicFrames will result nor how many productions will be applied to get a result. Backtracking is absolutely essential to avoid the combinatorial explosion that occurs when a complete set of boundary designations must be made before the result can be evaluated.
We conjecture that formally declaring the existence of ambiguity at the BNF level will result in large payback in terms of automating the process of building a system from its design. We perceive the primary gain lies in the coalescence of a potentially large set of special cases for handling the different outcomes into a single "invalid result - try again" state that can trap *all* unexpected outcomes. As the system becomes better able to select the proper behavior (in this example, correct boundaries), this state is entered less frequently. In the unambiguous case it is never entered and can be ignored. The power here devolves from using BNF fragments from the same toolbox to specify the operational behavior of two very different systems, datastream parsing on one hand and geometric decomposition on the other.

2.4 Ambiguous Situations

The following are examples where the very nature of the problem in question appears to be ambiguous from the point-of-view of the engineered problem solution:
(1) High-tech devices, e.g., photocopying machines, having one set of states associated with each 'normal' mode of operation, and a completely separate set of states for failure modes. Examples of failure modes include paper exhaustion, paper jams, mechanical failure, incorrect paper in the paper trays, and power interruption.

(2) In certain high-consequence operations involving data transfer, data processing must continue during and after periods of data loss. In a context-sensitive situation, the data lost might have altered the behavior of subsequent processing steps. In this event, assumptions must be made about the lost data so processing can proceed.

(3) Locomotive designs that manage potential energy independently from kinetic energy are being considered. Although accurate systems for controlling locomotive energy could be built, they would rest on assumptions about the actual system performance. Reacting to discrepancies between actual and predicted locomotive performance involves extremely complicated heuristics. Two of the more common concerns of railroad operators are the change in wheel/track friction as the wheels wear, for which a trend can be predicted, and the change in efficiency due to changes in air pressure, which varies in real time and whose value cannot be predicted.

(4) A huge majority of the traffic lights in the US are capable of operating on a network and acting adaptively. As of December 1996, officials at the California Transportation Department (CALTRANS) estimated that only about 5% of all traffic lights nation-wide were operating in any kind of adaptive mode.

There has been considerable research into systems that can deal with inconsistent or uncertain data. Some of these are constraint-based reasoning, case-based reasoning, backtracking, and variously controlled blind search-and-evaluate (for example, simulated annealing). The general strategy of *truth maintenance* appears to be a particularly fruitful approach because it enables proposals to be made and subsequently withdrawn (along with all their effects) if results are found wanting. A foundation work by deKleer (deKleer, 1985) is abstract, and implementations of it are largely inefficient and unpredictable for high consequence applications. Further, none of these approaches (nor any others, in general) generate domain-specific representations of ambiguity. Given the number of examples we have identified where ambiguous or inconsistent information flows would need to be handled, we conclude there is a need for:

(1) formal computation models based on some form of non-monotonic logic

(2) research into the structure of formal languages for specifying inconsistent and ambiguous data, and

(3) better tools for generating formal models based on (1) and (2).

3. Comparison

Any software engineering methodology is limited by its underlying processing model, in most cases a finite state machine. Current methods, although successful in simple real-time domains such as consumer electronics, are inadequate for large-scale systems, such as satellite ground stations or mission-critical control software. Sources of complication for such applications are their sheer size, uncertainty, and sensitivity of the algorithms to context. Even the technology being developed as part of the sophisticated DARPA program in Evolutionary Design of Complex Software appears to suffer from this same limitation.

We are extending the state of the art through modeling formalisms that can handle context-sensitive situations and algorithms that can deal with ambiguous data, by which we mean data with inconsistencies inherent in the methods by which it is acquired.

We intend to provide new methods of developing components by providing new computational models for the types of applications outlined above. The technology we are starting from has achieved some notoriety through demonstration in the Artificial Intelligence community, but has not achieved wide acceptance due to the academic nature of the implementations. We also expect to pursue the use of a formal specification language to form the basis of the interface between the application and the truth-maintenance technology.